Department of Computing, Imperial College London

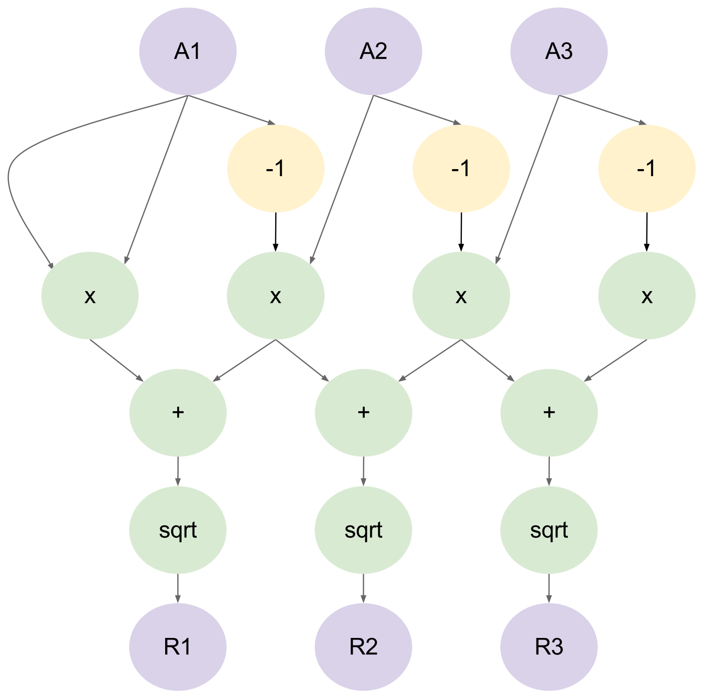
Custom Computing Research Group

Aspect Oriented Design for Dataflow Engines

– Synopsis –

Paul Grigoraş

paul.grigoras09@imperial.ac.uk



Traditional computing architectures process instructions sequentially, fetching them from computer memory, executing them and recording their results. This provides a convenient programming model but limits performance and energy efficiency. Transition to a dataflow architecture, where data are instead streamed through a static instruction graph can improve performance and energy efficiency by orders of magnitude, drastically reducing computation time in vital areas such as weather forecasting, oil and gas exploration and financial risk assessment. One key challenge is to improve the quality of dataflow programs and to enhance the productivity of dataflow programmers; it is what we aim to address in this project.

1 Introduction

Existing work shows that dataflow architectures can achieve significant performance gains compared to multi-core processors when implementing high throughput, highly parallel applications that operate on large, uniform data sets [1, 2]. However, due to poor developer productivity, the dataflow paradigm is not widely adopted and imperative languages are significantly more popular [3].

High-performance applications could benefit greatly from the adoption of dataflow computing since they require a significant amount of computational resources. By emulating dataflow machines on Field Programmable Gate Arrays (FPGAs) – programmable hardware chips – orders of magnitude increase in performance and energy efficiency can be achieved. For example, an implementation of Reverse Time Migration – an advanced seismic imaging application used by Chevron for oil and gas exploration – has been shown to be 103 times faster and 145 times more energy efficient than an optimised implementation running on a multi-core microprocessor [4]. Other important applications of dataflow computing include the recent deployment of a dataflow cluster at J.P. Morgan which reduced the computation time for complex risk assessment scenarios from hours to minutes.

In this project we propose a novel design-flow for High-Performance Computing (HPC) applications to support:

- **more efficient HPC** – by using FPGA based dataflow engines to achieve drastic increase in performance and energy efficiency;
- **more productive HPC** – by using *Aspect Oriented Programming* to decouple application optimisation from application development improving portability and maintainability.

2 Design Flow

To improve both efficiency and productivity, our design flow improves the *performance* and *energy efficiency* of existing applications while using a more systematic approach for design optimisation that results in more *portable* application code, improves *integration* with existing applications and *automates* manual, time-consuming and error-prone tasks.

The key components of the design flow are:

- *FAST*, a novel language for specifying dataflow designs which is compatible with C syntax, improving developer productivity and supporting combined hardware and software specifications;
- *aspect driven compilation flow*, used to decouple optimisation from design development, improving portability, and automating code generation and design space exploration [1], improving productivity;
- *systematic design space exploration*, to identify efficient configurations by using *aspect descriptions* to control the exploration process based on user-specified constraints.

[1]the process of exploring multiple configurations to identify optimal implementations

Figure 1 illustrates the design flow:

1. a C application containing an embedded high-level dataflow design specified in FAST is developed from the original source application;
2. the dataflow design is transformed according to the aspect descriptions in the repository to generate new configurations (e.g. with multiple word-length configurations);
3. the generated configurations are compiled using a backend compilation toolchain to FPGA dataflow designs;
4. feedback from the compilation process is used to drive design space exploration until user-specified constraints are satisfied.

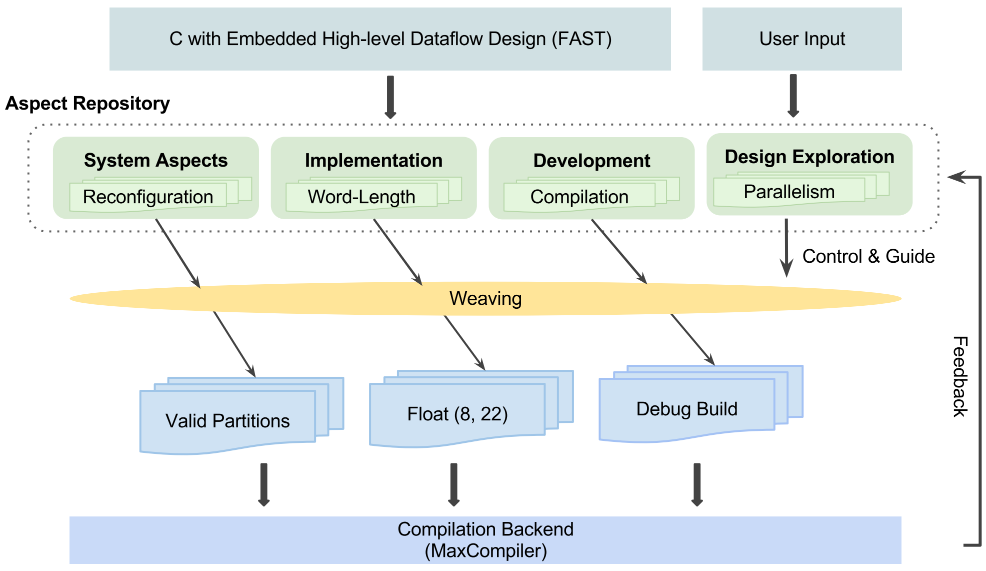


Figure 1: Proposed approach for aspect-driven compilation of dataflow designs.

3 Implementation

FAST (Facile Aspect-driven Source Transformation) is a novel language for specifying dataflow designs used as a starting point for the design flow described in Section 2. C syntax captures the dataflow computation and aspect descriptions specify the transformations required to optimise the dataflow implementation.

Listing 1 shows a simple implementation of a dataflow kernel for Black-Scholes option pricing, highlighting some of the most important features of FAST:

- dataflow kernels are declared as C functions with inputs and outputs defined in the function's signature;
- *streams* of data are declared and accessed as C pointers;
- stream "direction" (input or output) is inferred by the compiler;
- array indexing is used to access previous, current or past stream values;
- type width information is decoupled from the application code and can be added automatically via aspect generated pragma statements;

- C function calls are mapped to dataflow kernels via pragmas which provides the flexibility of selecting between software and dataflow implementations at run-time.

```
// 1. both input and output streams are declared in kernel header
// 2. no need for additional type definitions
void Price_FPGA(float* stockPrices, float *r,
                float c1, float c2, float c3,
                int nStocks, int stencilOrder, int timesteps)
{

  // 3. CURRENT CYCLE value used instead of counter API
  int stockstep  = (CURRENT_CYCLE / n1) \% timesteps;

  #pragma fast DSPBalance:full
  float result = stockPrices[0] * c1 + stockPrices[1] * c2 + stockPrices[-1] * c3;

  // 4. boolean types used for conditions
  bool up = (stockstep >= stencilOrder) && (stock < stockstep - stencilorder);

  // 5. assigning to output stream automatically outputs value
  r[0] = up ? result : stockPrices;

}

// Regular C style CPU implementation and main function
void Price_CPU(...) {...}

int main() {
  #pragma FAST hw:Price_FPGA
  Price_CPU(...);
}
```

Listing 1: FAST dataflow kernel for European Options pricing

Table 1 summarises the features of FAST and shows that many are implemented using C syntax. This is deliberate as it simplifies the programming model and facilitates direct translation of C applications to dataflow designs.

Feature	**Implementation**	**Method**	**Compared to C**
Dataflow Kernels	Function definition	C99	Cyclic execution
I/O	Kernel arguments	Inferred	Only params, =
Control	Ternary op. (?:), `if`	C99	
Computation	log, exp, sqrt, sin etc. +, *, /, -	math.h C99	Pragmas for range
Streams	Declared as pointers Array index access	C99	Cyclic update; negative offsets
Optimisation Hardware Mapping	C pragmas	C99	Run-time values
Parameterization	Constants, variables, `for`, `while`	C99	Compile-time bounds

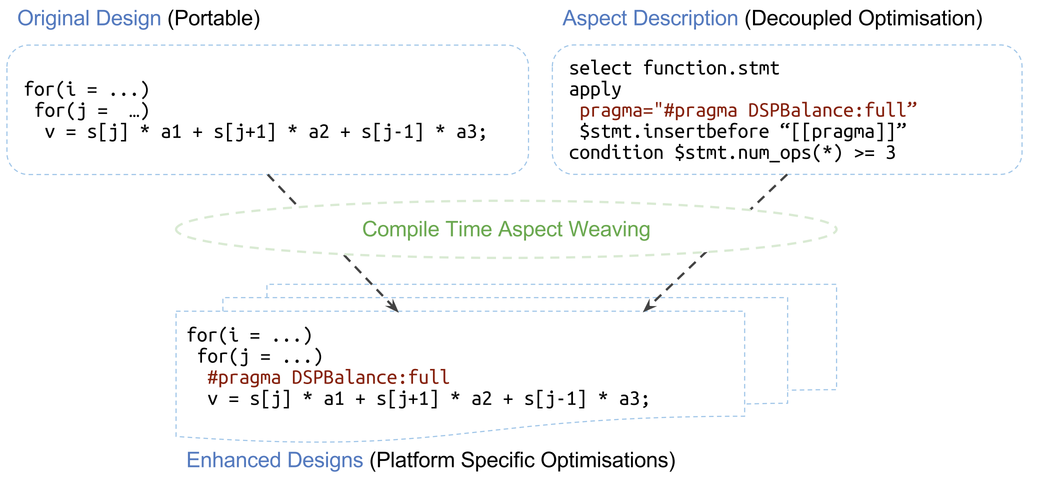
Table 1: Summary of FAST features.

FAST dataflow designs are compiled by the `fastc` compiler to MaxCompiler [5] designs, a commercial tool for dataflow engines. MaxCompiler adopts a low-level dataflow language, so by providing a higher-level programming interface, we improve productivity of dataflow programmers.

`fastc` extends the ROSE [6] compiler framework to produce an Abstract Syntax Tree (AST) from C sources. `fastc` comprises 47 classes, including a decoupled library for analysis of dataflow graphs, totalling approximately three thousand lines of executable code and two thousand lines of tests. `fastc` transforms the ROSE AST in a series of compiler passes including:

1. **Extract dataflow kernels** – separates FAST dataflow kernels from the rest of the source C application;
2. **Constant Extraction** – extracts design constants from source files. These are global constants used to parameterise the FAST dataflow designs and can be shared with the CPU code;
3. **Pragma Extraction** – the pragma extraction pass analyses and extracts information from pragmas that specify types of streams, ranges of offset streams and other design optimisations;
4. **Infer input and output streams** – infers stream direction, as follows:
   (a) extract kernel parameter set, P
   (b) extract pointer parameters which form the stream set, $S \subset P$
   (c) perform a written analysis and record the kernel modifies set, M
   (d) compute the set of output streams, $O = M \cap S$
   (e) compute the set of inputs streams, $I = P - O$
   (f) detach assignments to output streams from the original AST to prevent traversal on future passes and add corresponding output (and input) nodes to the dataflow design
5. **Inline auxiliary functions** and **Static Single Assignment Renaming** – inline calls to helper functions made from FAST dataflow kernels;
6. **Dataflow Graph Generation** – traverse the AST of every dataflow kernel to create a corresponding dataflow graph;
7. **MaxCompiler Design Generation** – traverse the constructed dataflow graph and the original AST to generate corresponding MaxCompiler designs for extracted dataflow kernels.

However, since FAST only captures the basic elements of a dataflow design, *aspect descriptions* are used to augment our flow with additional capabilities. Aspect descriptions are modules that capture cross-cutting concerns which can be decoupled from the primary function of a program. Through *weaving* (Figure 2) these modules can be combined with application code to allow the addition of new functionality.


Original Design (Portable)
Aspect Description (Decoupled Optimisation)
Compile Time Aspect Weaving
Enhanced Designs (Platform Specific Optimisations)

```
for(i = ...)
 for(j =  …)
  v = s[j] * a1 + s[j+1] * a2 + s[j-1] * a3;
```

```
select function.stmt
apply
 pragma="#pragma DSPBalance:full”
 $stmt.insertbefore “[[pragma]]”
condition $stmt.num_ops(*) >= 3
```

```
for(i = ...)
 for(j = ...)
  #pragma DSPBalance:full
  v = s[j] * a1 + s[j+1] * a2 + s[j-1] * a3;
```

Figure 2: Aspect weaving overview.

In this project we combine FAST dataflow designs with the LARA aspect-oriented design-flow [7] that performs weaving at compile-time to generate new sources which incorporate both functional elements of the original sources, and non-functional concerns (such as optimisations) captured by the aspect descriptions. This approach makes the functionality of the application easier to understand – since computation is expressed more intuitively in FAST – and results in more maintainable and portable designs that are not obscured by algorithmic transformations or platform-specific optimisations. Additionally, strategies can be re-used across different applications, improving developer productivity.

We group aspect descriptions used with our approach in four main classes: *i. system aspect descriptions*, are applied at the whole system level (FAST + C application) to capture the mapping between application modules and dataflow accelerator, *ii. implementation aspect descriptions*, operate at the level of the dataflow design, to apply platform specific optimisations, *iii. exploration aspect descriptions*, are used to explore the design space of optimisation trade-offs and, finally, *iv. development aspect descriptions*, support the development process (e.g. by facilitating debugging).

An example of an implementation aspect description is the operator optimisation aspect of Listing 2. This aspect description maps operators in the program to dedicated hardware resources called Digital Signal Processors (DSPs), by inserting an optimisation pragma (`#pragma FAST balanceDSP:<factor>`) before statements that match a certain operator usage pattern, provided as input by the user (e.g. three multiplications and two additions).

```
aspectdef DspBalancing
input: opMapping
function.stmt:
  opUsage = countOperatorUsage(stmt)
  factor  = opMapping[opUsage]
  if (factor != '')
    stmt.prepend('#pragma␣fast␣balanceDSP:' + factor);
end
```

Listing 2: Operator optimisation aspect description.

Then, the aspect description in Listing 3 can be used to explore the design space of operator optimisations, thus automating a common design optimisation pattern: vary an attribute of a dataflow configuration, generate the FAST configuration and corresponding FPGA design, extract feedback from the compilation report and stop when a resource usage limit is reached.

```
aspectdef DesignExploration
input: attribute, start, step, res, res_limit, config
  config[attribute] = start
  do {
    var designName = genName(config)
    generateFASTDesign(designName, config)
    buildFASTDesign(designName)
    config[attribute] += step
  } while (@hw[designName].res < res_limit)
end
```

Listing 3: Iterative design space exploration aspect description.

4 Evaluation

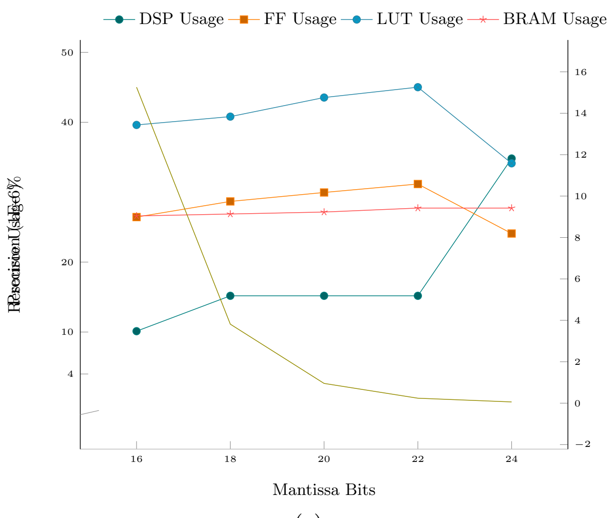
We evaluated our approach by implementing a variety of real-life applications including:

- *Numerical Differentiation* with the Savitsky-Golay smoothing filter, a scientific application used to compute the derivative of a function when its analytical form is not available (e.g. on large experimental data);
- *Black Scholes Option Pricing*, a financial application for pricing European options[2];

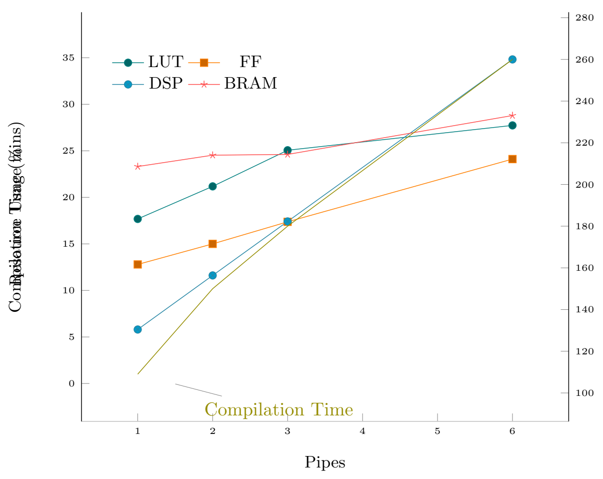
[2]financial instruments that allow the owner to buy or sell the underlying asset at a fixed point in the future

- *Ad Prediction*, a Bayesian inference application for predicting the rate at which ads are clicked on by users of Microsoft Bing.

However, due to lack of space, we focus on *Reverse Time Migration* (RTM), the most complex application in our suite. RTM is a seismic imaging application used to detect geological structures, based on the Earth's response to injected acoustic waves. We used FAST to implement the dataflow kernels for the memory controllers and the computational kernel. We used the design space exploration aspect description of Listing 3 to analyse trade-offs between accuracy and resource usage (Figure 3.a) and identify resource bottlenecks (Figure 3.b). This shows that simple aspect descriptions can be used effectively to identify counter-intuitive results (e.g. large savings in DSP usage achieved when decreasing mantissa from 24 to 22 bits). The automated process also improves design portability, allowing optimisations to be explored on different platforms without manual intervention.



(a)



(b)

Figure 3: *(a)* Exploration of accuracy versus resource usage trade-offs using the aspect description of Listing 3 and *(b)* increase in compilation time and resource usage with number of parallel cores, used to identify design bottlenecks (limited resources which negatively impact design scalability).

Section A, Table 2 compares FAST implementations with other published results, showing that we significantly outperform CPU implementations in terms of energy efficiency (by a factor of 147.1) and performance (by a factor of 101.3) and match the performance of state-of-the-art manually implemented FPGA designs. Other results from our benchmark suite (summarised in Section A, Table 3) show that we can significantly reduce the number of API calls (by a factor of up to 16) and lines of code required (by up to 60%) while maintaining performance when compared to manually developed dataflow designs for a wider range of applications.

5 Conclusion

We have tackled the challenge of improving developer productivity with enhanced performance and energy efficiency for custom dataflow designs implemented on FPGAs, showing that this can be achieved by adopting a novel design flow based on the aspect-oriented philosophy of encapsulating cross-cutting concerns in highly cohesive aspect descriptions. The project has been included in the FP7[3] funded HARNESS Project where it would be used for generating efficient dataflow implementations for key cloud applications. A full paper based on this project was accepted for publication at the 24th IEEE International Conference on Application-Specific Systems, Architectures and Processors, ASAP 2013 [8].

[3]European Union Seventh Framework Programme

A Experimental Data

	CPU	**GPU** [9]	**GPU** [10]	**GPU** [11]	**FPGA** [12]	**FPGA** [4]	**FPGA**	**FPGA D**^{4} [4]	**FPGA D**
Freq.(GHz)	2.7	1.15	1.15	1.15	0.1	0.1	0.1	0.1	0.1
Time (s)	1458	52	–	–	–	18	18.3	13.4	13.6
GFLOP/s	0.9	51.2	36	64.5	35.8	68.0	66.8	91.6	90.2
Speed-up(X)	1	56.8	40	71.6	39.7	76.4	**74.22**	102.9	**101.3**
Power (W)	185	–	461	–	–	129	126	128	125
Efficiency	4.9	–	78	–	–	527.1	527.7	715.6	721.6
Eff. Gains	1	–	15.9	–	–	107.5	**107.0**	146.0	**147.1**

Table 2: Comparison of our RTM implementation (results highlighted in bold) with other published results for CPU, GPU and FPGA. FPGA implementations marked *D* (Dynamic) exploit the possibility to reconfigure the FPGA chips with a new dataflow design at run-time to remove idle functions. This leads to improved performance and energy efficiency.

Kernel	**Size**	**Reduction**	**API Calls**	**Performance**	**Resource**
MemoryRead	48	43 %	4.33	$> 75\%$	$\approx 100\%$
MemoryWrite	54	31 %	4.13		
RTMCompute	231	27 %	10	$\approx 100\%$	
SmoothFilter	60	45 %	14		
Differentiation	58	42 %	14		
Black-Scholes	56	60 %	5.5		
AdPrediction	94	40 %	16		$\approx 117\%$

Table 3: Comparison of manual MaxCompiler designs and FAST designs. Column 2 gives the size of the benchmark in executable lines of code, while columns 3 and 4 measure the reduction in code size and API calls achieved by the FAST implementations, showing that code size can be reduced by as much as 60% while using as much as 16 times less API calls. MemoryRead and MemoryWrite are the memory controller kernels and RTMCompute is the computational kernel of our RTM implementation. SmoothFilter and Differentiation implement the filtering and differentiation of our numerical differentiation implementation. Black-Scholes and AdPrediction represent the computational kernels for Black-Scholes options pricing and Bing Ad Prediction respectively.

References

[1] M. Flynn, O. Pell, and O. Mencer, "Dataflow Supercomputing," in *FPL*, 2012.

[2] O. Mencer, "Maximum Performance Computing for Exascale Applications," in *SAMOS*, 2012.

[3] Tiobe Software, "Tiobe Programming Index," http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, 2012.

[4] X. Niu, Q. Jin, W. Luk, Q. Liu, and O. Pell, "Exploiting Run-Time Reconfiguration in Stencil Computation," in *FPL*, 2012.

[5] O. Lindtjorn, R. G. Clapp, O. Pell, O. Mencer, M. J. Flynn, and H. Fu, "Beyond Traditional Microprocessors for Geoscience High-Performance Computing Applications," *Micro, IEEE*, vol. 31, no. 2, pp. 41–49, 2011.

[6] D. Quinlan, "ROSE: Compiler Support For Object-Oriented Frameworks," *Parallel Processing Letters*, vol. 10, pp. 215–226, 2000.

[7] J. M. Cardoso, T. Carvalho, J. G. Coutinho, W. Luk, R. Nobre, P. Diniz, and Z. Petrov, "LARA: an Aspect-Oriented Programming Language for Embedded Systems," in *AOSD*, 2012.

[8] P. Grigoras, X. Niu, J. Coutinho, W. Luk, J. Bower, and O. Pell, "Aspect Driven Compilation for Dataflow Designs," in *ASAP*, 2013.

[9] E. H. Phillips and M. Fatica, "Implementing the Himeno Benchmark with CUDA on GPU clusters," in *IPDPS*, 2010, pp. 1–10.

[10] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil Computation Optimization and Auto-Tuning on State-of-the-Art Multicore Architectures," in *SC*. IEEE Press, 2008, p. 4.

[11] Y. Yang, H.-M. Cui, X.-B. Feng, and J.-L. Xue, "A hybrid circular queue method for iterative stencil computations on gpus," *Journal of Computer Science and Technology*, vol. 27, no. 1, pp. 57–74, 2012.

[12] M. Araya-Polo *et al*, "Assessing Accelerator-based HPC Reverse Time Migration," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 147–162, 2011.